Noname manuscript No.
(will be inserted by the editor)

Static and Dynamic Data Models for the Storage Resource Manager v2.2

Andrea Domenici · **Flavia Donno**

Abstract We present a conceptual model for the Storage Resource Manager, the standard interface adopted for the storage systems of the Worldwide LHC Computing Grid. This model provides a clear and concise definition of the structural and behavioral concepts underlying the interface specification and is meant to support service and application development. Different languages (natural language, UML diagrams, and simple set-theoretic and logical notation) are used to describe different aspects of the model.

Keywords Grids, storage, specification, UML

1 Introduction

The High Energy Physics community at CERN and several other laboratories will use the data acquired by the Large Hadron Collider (LHC) to explore the fundamental laws of the Universe. Several (10–15) Petabytes of data will be collected by the 4 experiment detectors every year. The computational and storage facilities needed to process the data will be provided by the Worldwide LHC Computing Grid (WLCG), one of the largest Grid infrastructures dedicated to high-performance scientific computation, counting today more than 200 sites all over the world.

The WLCG must provide, among other facilities, a Grid storage service featuring several storage management functions, including dynamic space allocation, the negotiation of file access protocols, support for quality of storage, authentication and authorization mechanisms, storage and file management, scheduling of space and file operations, and support for temporary files. This service relies on a few different types of high-end storage systems deployed at many Grid sites.

Andrea Domenici
DIIEIT, University of Pisa, v. Diotisalvi 2, I-56122 Pisa, Italy

Flavia Donno
CERN, European Organization for Nuclear Research, CH-1211 Geneva 23, Switzerland
Tel.: +41 76 48 753 98
Fax: +41 22 76 765 55
E-mail: Flavia.Donno@cern.ch

In order to provide uniform, site-independent access to the service, a standard interface, called the *Storage Resource Manager* (SRM) was proposed [13].

The WLCG has coordinated an international collaboration that has produced the *SRM v2.2 Interface Specification* [14], submitted to the OGF as a proposed RFC. The SRM v2.2 is currently in production service in the WLCG infrastructure and is being used in data management and access exercises in view of the upcoming data acquisition when the LHC begins operating.

This interface is specified primarily as an application programming interface (API), i.e., a set of requests whereby an application may obtain the desired storage management services. These requests imply a structural and behavioral model of the SRM that is an abstraction of the actual implementations of the interface.

In this paper we propose a conceptual model for the SRM that should supplement the API and other specifications with an explicit, clear and concise definition of its underlying structural and behavioral concepts. This model has been used in the definition of the service semantics and has supported a more rigorous validation of its implementations. In the future, it can be a tool for service development.

The proposed model addresses both service and application developers, and it is meant to strike a satisfactory compromise between clarity and formality. Different notations (e.g., basic set-theoretic and logical formalism, UML [11] diagrams, and plain English) are used as appropriate to define different aspects of the model.

This paper is an amply extended version of the one presented at the ISGC 2007 Conference [6].

2 Storage elements

A *Storage Element* (SE) is a Grid Service implemented on a mass storage system (MSS) that may be based on a pool of disk servers, on more specialized high-performing disk-based hardware, or on a disk cache front-end backed by a tape system, or some other reliable, long-term storage medium. Remote data access is provided by a GridFTP service [1] and possibly by other data transfer services, while local (intra-cluster) access is provided by POSIX-like input/output calls. Authentication, authorization and audit/accounting facilities are also part of a SE.

A Storage Element provides *spaces* where users create and access *files*. A file is a logical set of data that is embodied in one or more physical *copies* (much like a book is a logical piece of literature that is embodied in the copies sold in bookstores).

Storage spaces may be of different qualities, related to reliability and accessibility, and support different data transfer protocols. Different users may have different requirements on space quality and access protocol, therefore, in addition to the basic data transfer and file access functions, a SE must support more advanced resource management services, including dynamic space allocation. Clients must be able to access the resource management services through a standard interface, independent of the underlying MSS. This interface is offered by the Storage Resource Manager.

3 The Storage Resource Manager interface

The *Storage Resource Manager* (SRM) is a middleware component whose function is to provide dynamic space allocation and file management on shared storage components on

the Grid. More precisely, the SRM is a Grid service associated with a Storage Element, with several different implementations, each targeted to a specific Mass Storage System. Its interface is defined by the *SRM Interface Specification* [14] that lists the service requests that a client application may issue, along with the data types for their arguments and results.

Request signatures are given in an implementation-independent language and grouped by functionality:

- *Space management* requests allow the client to reserve, release, and manage spaces, specifying or negotiating their qualitiy and lifetime.
- *Data transfer* requests have the purpose of getting files into SRM spaces either from the client's space or from other remote storage systems on the Grid, and to retrieve them.
- *Directory* requests create, populate, list, or delete directories.
- *Permission* requests set or list read and write permissions on files and directories.
- *Discovery* functions allow applications to query the availability and characteristics of the storage system behind the SRM interface.

We will mention only the requests that will be referred to in the rest of the paper. Among them, the following space management requests:

srmReserveSpace allows the requester to allocate space with specified properties.
srmReleaseSpace releases an occupied space. If the space contains copies of a file, the system must check if those copies can be deleted.
srmChangeSpaceForFiles is used to change the space where the files are stored.
srmExtendFileLifeTimeInSpace is used to extend the lifetime of files that have a copy in the space.

And then we list the following data transfer requests:

srmPrepareToPut creates a *handle*, i.e., a reference that clients can use to create new files in a storage space or overwrite existing ones.
srmPutDone tells the SRM that the write operations are done.
srmCopy allows an SRM server to copy a local file to another SRM server or to retrieve a file from it.
srmBringOnline is used to make files ready for future use. The system may stage copies from a slow medium such as a tape system to a faster one such as a disk.
srmPrepareToGet returns a handle to an online copy of the requested file.
srmReleaseFiles marks as releasable the copies generated by srmPrepareToGet or srmBringOnline.
srmAbortRequest, srmAbortFiles force termination of asynchronous requests.
srmExtendFileLifeTime extends the (pin) lifetime of files, copies, or handles.

4 A model for the Storage Resource Manager

The main specification documents for the SRM are the above mentioned Interface Specification and the *Storage Element Model for SRM 2.2 and GLUE schema description* [2]. Other relevant documents are [13, 12, 7].

The Interface Specification has the purpose of defining the SRM API, therefore it is not meant to provide an overall view of the underlying concepts, while the GLUE schema is a UML model meant to define only the SRM properties relevant for the Information Service, so it cannot fully represent the SRM and particularly its behavior. Initially, the work on the SRM was focused on interface definition and implementation, but now the

SRM development has reached a stage where an explicit model of the service semantics is useful for interface designers, service developers, testers, and users.

This model should be a synthetic description of a user's view of the service, with the basic entities (such as *space*, *file*...), their relationships, and the changes they may go through.

We have chosen to use two models, with different levels of formality. The semi-formal model uses plain English and UML diagrams, and it is meant to give an overall view of the system, identifying its main components, their relationships and behavior, and to define and clarify the terms that users and developers read in the Interface Specification. A more formal model uses the well-known set-theoretic and logical notation to express constraints. This model is meant to resolve ambiguities that might remain in the semi-formal model, and to support the design and testing of SRM implementations. Note that not even this model is completely formal, since several details have been left out: For example, the concept of *time* has not been formalized. A complete axiomatization would be needed in order to use some automatic proof system, but usage of such sophisticated methods is left for future development, after the initial version of the model has been validated by its users.

4.1 Describing concepts and properties

The SRM model has been built according to the classical object-oriented approach: The most relevant concepts have been identified and represented as object classes, their properties and reciprocal relationshisps being modeled by attributes and associations subject to various constraints. All this information constitutes the *static model*.

When the SRM Interface Specification document and the GLUE Schema are considered, some basic concepts, such as *space* and *file*, are immediately evident. However, finding the best way of characterizing them is not quite obvious. The documents provide informal definitions for them, but in order to define their attributes it was often necessary to go through the Interface Specification and see what input parameters in the API affect the SRM behavior with respect to the given entities.

In the following we define the concepts that have been deemed necessary to model the SRM and we describe their attributes and associations, possibly with related constraints. Fig. 1 shows a partial UML class diagram for the SRM static model, where the Storage Element has been left out.

In the diagram, the attribute names are usually related to the names of parameters (or fields of structured parameters) occurring in the request signatures published in the Interface Specification. The attribute types are the types defined in the Interface Specification.

4.1.1 Storage Class and File Storage Type

The execution of most SRM operations depends on a few properties of the involved files and of the spaces where their copies reside. In this section we introduce the properties of *storage class* and *(file) storage type*. The former is defined in terms of two other properties, *retention policy* and *access latency*.

The properties of retention policy and access latency may or must be specified for most SRM requests involving the reservation or creation of spaces and files. Retention policy describes the reliability of a storage medium, while access latency says if data are immediately accessible or must be staged from a slow medium (e.g., tape) to a faster one.

Retention policy is a qualitative indication of the likelihood that a file copy may be lost in a given storage space. This likelihood may be high, intermediate, or low. A space with a

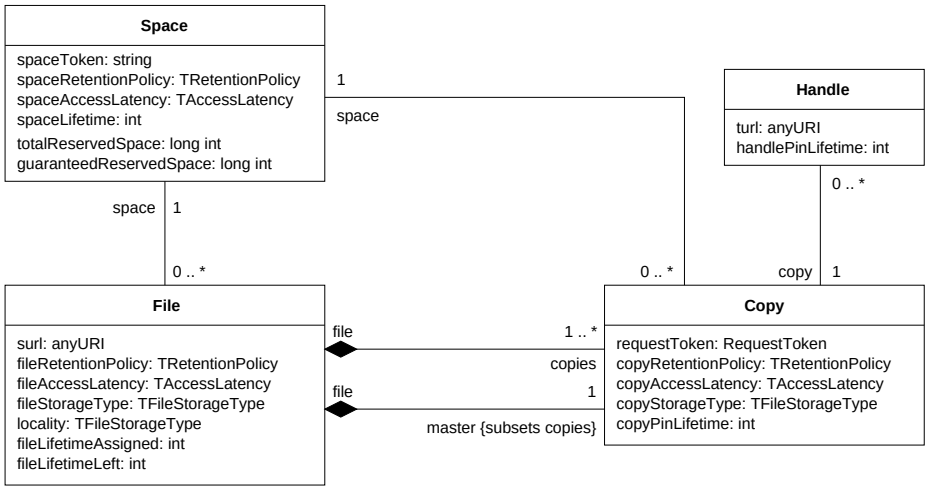

Fig. 1 Static model of the SRM.

Table 1 Storage classes

Retention policy	Access latency	
	ONLINE	NEARLINE
REPLICA	Tape0Disk1	Tape0Disk0
CUSTODIAL	Tape1Disk1	Tape1Disk0

high likelihood of file loss is said to have a REPLICA retention policy, since it is satisfactory for replicated files that can be accessed with a limited performance penalty if a single copy is lost. A space with an intermediate likelihood of file loss has an OUTPUT retention policy, since it is satisfactory for files that are not replicated but can be recreated as the output of some computation. Finally, the CUSTODIAL retention policy applies to storage that has a low likelihood of file loss, and is therefore appropriate for files whose recovery would be very costly or even impossible.

Access latency is a classification of storage media according to the timeliness of data access. A space where data are immediately accessible is ONLINE, otherwise it is NEARLINE. A NEARLINE space is supported by a medium, such as tape or DVD libraries, that uses mechanical operations (beyond disk spinning) to retrieve the data, that are then staged to temporary disk storage. The SRM specification considers also an OFFLINE category of access latency, for data requiring manual intervention to be made accessible, but this category is not supported by the current SRM implementations and will be ignored in the following. Also the OUTPUT value of retention policy is currently unsupported, and therefore ignored.

The possible combinations of these properties supported within the WLCG are shown in Table 4.1.1. In WLCG such combinations are referred to as *storage classes* and called *Tape0Disk0*, *Tape0Disk1*, etc.

While the storage class is a property of spaces, the *storage type* is a property of files that refers to their lifecycle. A VOLATILE file has a limited lifetime, and it is deleted after the lifetime has expired. A DURABLE file also has a limited lifetime, but the SRM may not delete it automatically, instead it must be removed explicitly by the file owner. A PER-

MANENT file has an unlimited lifetime, and may be removed explicitly by the file owner. Durable files are not supported by the current WLCG implementations and will be ignored in the following.

4.1.2 Storage Element

A *storage element* can be seen as an aggregate of storage media, possibly with different charcteristics: for example, a single storage element might have both tapes and disks, or disks of different kinds. Further, different media might be accessed through diferent protocols (such as GridFTP or RFIO). The characteristics of the storage media determine the storage classes of the allocated spaces. The main properties of the storage element are then its *identifier*, the *size* of the available storage, the supported storage classes, and the supported *protocols*.

To save space, the UML class modeling the storage element is not shown in Fig. 1. In the complete diagram, it is linked to the *Space* class through a composition (i.e., strong ownership) relationship.

4.1.3 Space

A *space* is a part of a storage medium that can be reserved for a user. Users reserve space of a given requested size with a given storage class, i.e., with given values of retention policy and access latency, and then may store files on them. Space reservation works on a best-effort basis: If the user asks for a given amount of space to be reserved, the system might respond with a smaller amount of reserved space that the user can decide to release if insufficient.

The value of attribute *totalReservedSpace* may then be different from the requested size. Further, the availability of this amount of space is not guaranteed. The user may request a guaranteed amount of space, and the value of attribute *guaranteedReservedSpace* is the amount of space actually guaranteed.

Each space is uniquely identified by a *space token*, represented as an opaque string returned by the space reservation operation.

Finally, a space has a lifetime, limited or unlimited.

4.1.4 File

A *file* is a logical dataset that resides on a space. More precisely, a file has one or more physical instances, or *copies*. One of the copies is the *master*, or *primary*, copy, and a file is said to *reside* in the space containing the master copy.

A file is identified by a *Site URL* (SURL), a string of the form:

$$\texttt{srm://}\mathit{host}[\texttt{:}\mathit{port}]/[\mathit{soap\_end\_point\_path}\texttt{?SFN=}]\mathit{site\_file\_name}$$

where the parts in square brackets are optional.

The *locality* of a file describes where its copies reside: the copies of an ONLINE file are all in ONLINE spaces, those of a NEARLINE file are in NEARLINE spaces, while an ONLINENEARLINE file has copies in both kinds of spaces. The UNAVAILABLE and LOST values of locality mean that a file's copies are temporarily or, respectively, permanently unavailable due to hardware failures.

A file has a storage type (see above) and also a retention policy and access latency. These latter properties, requested by the user when the file is created, specify a constraint on the

corresponding properties of the space containing the master copy: the space must have a better or equal value of retention policy and access latency than the specified file properties.

Volatile (and durable) files are initially assigned a lifetime (*fileLifetimeAssigned*). The value of attribute *fileLifetimeLeft* is the remaining lifetime at the time when a user request is serviced, and its value is returned by several SRM calls.

4.1.5 Copy

Most user requests referring to files (through their SURLs) affect their copies, but the copies themselves are not managed directly by the user. Instead, they are controlled by the SRM according to its space management strategies, compatibly with the space and file properties. The SRM distinguishes among the copies of a given file through the *request token*, an opaque string associated with the user request that caused the copy to be created.

Copies, too, have a retention policy, an access latency, and a storage type. Their retention policy and access latency are related to the space storage class with the same constraints seen for files. The storage type may be different from the storage tipe of the file.

A copy may be *pinned*, i.e., it may be guaranteed to be kept in a space for a given time, the *pin lifetime*.

4.1.6 Handle

Users do not know the location of a file's copies within a storage element, so that the SRM be free to create, move, and destroy copies according to its space management policies. When a user need to access the data, the SRM returns the needed information, consisting in the physical location of a copy in the storage element, and in the protocol to be used for data access. This information is encoded in a *Transport URL* (TURL), a kind of URI of the form:

$$protocol://host[:port]/physical\_file\_name$$

where the parts in square brackets are optional.

The TURL is valid only for a given timespan. A TURL and its validity timespan make a *handle* for an accessible copy. The value of the *handlePinTime* attribute is the validity timespan of the TURL.

4.2 Describing behavior

The static model is a vocabulary that defines the components of the SRM, with their properties and relationships. We can now refer to this vocabulary to describe the behavior of the SRM, i.e., its *dynamic model*. This behavioral description is arguably the most relevant contribution of the model for application developers, since it enables them to ascertain what sequences of requests are allowed by the SRM and what responses are expected. The dynamic model can then be used as a *protocol* for the SRM service.

We describe the dynamic model with the *Statecharts* formalism [8, 11] adopted in the UML. In this formalism, a state machine has a hierarchical structure, i.e., any of its states can be decomposed into substates, or, conversely, states can be composed into superstates.

Transitions are labeled by *triggers*, i.e., events that enable the transition, and possibly by *guards*, i.e., conditions that must hold for the enabled transition to take place. Trigger events may be occurrences of requests, denoted by the request name and possibly by request arguments, or time events, denoted by *when* clauses. An unlabeled transition is triggered by

the completion of the activity carried out by the system in the source state. In the diagrams, we have dropped the “srm” prefix from the names of requests.

Some conventions allow for a compact graphical representation: drawing a transition originating from the border of a superstate icon is equivalent to drawing transitions originating from each of the substate icons, with the same target and the same trigger and guard.

4.2.1 File behavior

A file is created with a *prepareToPut* or a *copy* request (Fig. 2). A request may involve several files and can be served asynchronously, so any file being created may remain for some time in a waiting state (*SURL_Unassigned*) before it is assigned a SURL. In this state, the file can be destroyed by an *abortFiles* or *abortRequest* operation. Otherwise, after some time a SURL is assigned and the file enters a state (*SURL_Assigned*) where further requests may affect it.

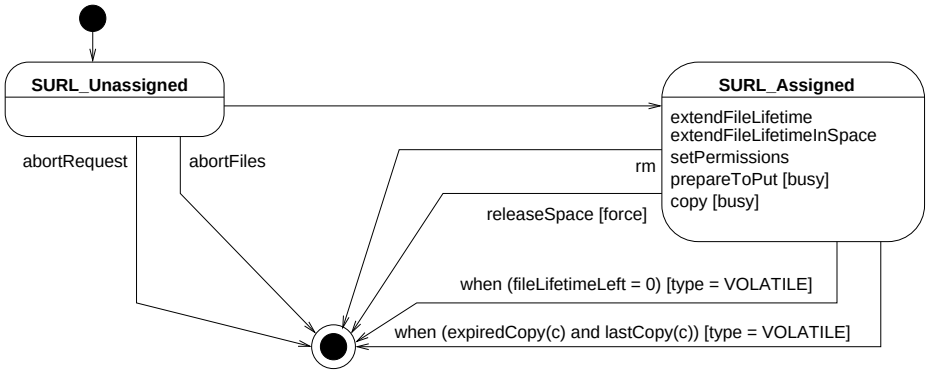

Fig. 2 State machine for File (1).

A file in the *SURL_Assigned* state can be destroyed by an *rm* (remove) request, by a *releaseSpace* request with the *force* (short for *forceFileRelease*) option, when its lifetime expires and the file is volatile, or when the pin lifetime of its last copy expires and the file is volatile.

Some requests are accepted in the *SURL_Assigned* state, but they do not alter the behavior. Such requests are listed as *internal transitions* (shown inside the state icon in the diagram) and leave the file in *SURL_Assigned* and in its current substate, whichever it be.

Other requests do change the file state, but still keeping it in the *SURL_Assigned* superstate. The evolution of a file in this state is shown in Fig. 3. Please note that the SRM standard allows implementations to exhibit a different behavior from the one shown here. Namely, after a *putDone* request for a CUSTODIAL file, an online copy may be kept at least temporarily, so that the file goes to the *NearlineOnline* state instead of the *Nearline* state.

When the SURL is first assigned, the file enters the *Busy* state. In this state, data can be transferred to the storage element by an external application (e.g., GridFTP). When the transfer is complete, the user notifies the SRM by issuing a *putDone* request. The next state depends on the retention policy requested for the file: if it is CUSTODIAL, the master copy is created in a NEARLINE space, and the file is, accordingly, in the *Nearline* state. This

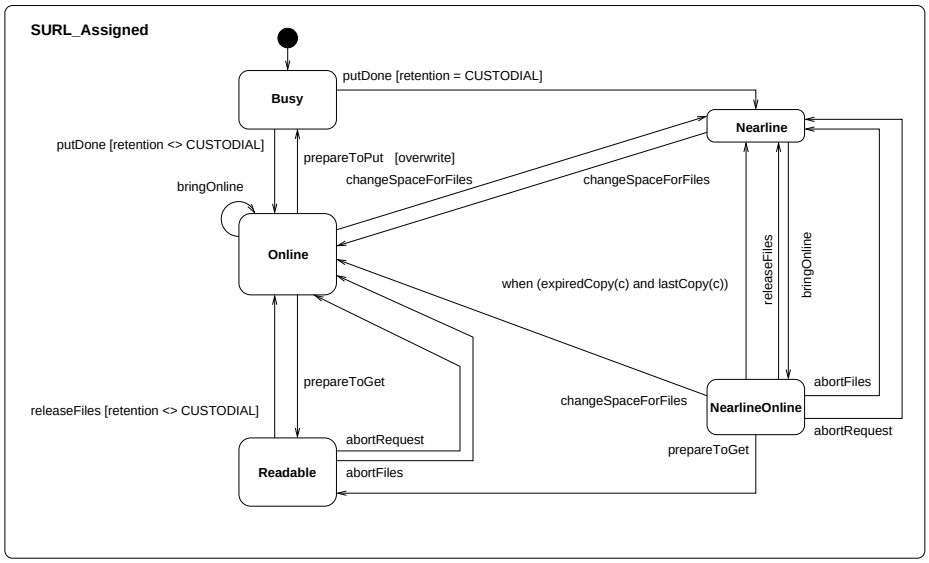

Fig. 3 State machine for File (2).

state then models a situation where all copies (typically, just the master copy) of the file are in NEARLINE space.

Copies of a file with CUSTODIAL retention policy may also be created in an ONLINE space, and the file is then in the *NearlineOnline* state. Some transitions may take it back to *Nearline*; in particular, the expiration of the pin lifetime of the last online copy.

If the retention policy is not CUSTODIAL, the master copy is created in an ONLINE space. When all copies of a file are in an ONLINE space, the file is in the *Online* state, where it may remain until expiration of its lifetime.

5 A more formal static model

In the preceding section a model of the SRM has been described in plain English. This informal description makes explicit many assumptions underlying the Interface Specification and introduces (hopefully with appropriate clarity) the relevant concepts. UML diagrams summarize the model in a synthetic, semi-formal manner.

While we feel that this informal and semi-formal representation is valuable for users and developers of the SRM, a finer level of detail and a greater degree of formality are needed to ensure interoperability and full compliance with the specification. Therefore we propose an initial, still incomplete, formal model expressed in basic mathematical notation. Since the SRM is still evolving and several issues are still being discussed among its developers, the model is limited to fundamental features, upon which further extensions and refinements can be built.

An elementary mathematical notation was chosen instead of some more specialized language, as most developers are familiar with standard set-theoretic and logical notation, while probably only a few of them have a working knowledge of formal specification languages. Otherwise, such languages as the UML *Object Constraint Language* [15] or the *Z Specifi-*

cation Language [16] would be attractive candidates. If the SRM community will consider adopting such languages, it should be easy to translate from the notation adopted here.

5.1 Basic definitions

In order to describe the SRM in term of elementary mathematical concepts, we first introduce some *basic sets* whose members are unstructured values, such as atomic symbols (meant to represent names of objects or discrete values for their attributes) or numbers. Then we define the constructed sets of *storage elements*, *spaces*, *copies*, *handles*, and *files* as Cartesian products of some previouly defined sets. Hence, each element of one of these constructed sets is a tuple with named components. We call the components *attributes*, and we use the dot notation to refer to the value of an object's component. For example, if a set S is defined as $B_1 \times B_2$, its elements are tuples of the form $\langle attr1, attr2 \rangle$, with $attr1 \in B_1$ and $attr2 \in B_2$. If an object o belongs to S, the expression $o.attr1$ denotes the value of its first component.

All sets defined in the following are mutually disjoined.

Only the most relevant attributes will be considered in this formalization. These attributes usually match the attributes introduced in the UML model, but in some cases a function replaces a UML attribute.

The sets corresponding to attributes will be introduced incrementally, i.e., when defining a new constructed set we will mention only the constituent sets that have not been previously introduced.

5.1.1 Common properties

We define the following basic sets:

Sizes	Sz	=	$\mathbb{N}$
Lifetimes	L	=	$\mathbb{N} \cup \{\top\}$
Retention policy	Rp	=	$\{\mathsf{replica}, \mathsf{output}, \mathsf{custodial}\}$
Access latency	Al	=	$\{\mathsf{online}, \mathsf{nearline}\}$

For set L, we have:

$$\forall_{t \in L}\ 0 \leq t \leq \top$$

where '$\top$' (*top*) denotes an unlimited value.
For sets Rp and Al we have, respectively:

$$\mathsf{replica} < \mathsf{output} < \mathsf{custodial}$$

$$\mathsf{online} < \mathsf{nearline} \ .$$

We define the set of *storage classes* as:

$$Sc = Rp \times Al$$

and a *storage class* as a tuple of the form

$$\langle retpol, latency \rangle \ .$$

5.1.2 Storage element

We define the following basic sets:

Storage element identifiers	$SEid$		a countable set of symbols
Protocols	P	=	$\{\mathsf{rfio}, \mathsf{dcap}, \mathsf{gsiftp}, \mathsf{file}\}$

We define the set of *storage properties* as:

$$Prop = Sc \times P$$

and the *storage element properties* as a tuple of the form

$$\langle sclass, protocol \rangle \quad .$$

The set of *supported properties* is the powerset (i.e., the set of subsets) of the storage properties:

$$Sprop = \mathscr{P} Prop \quad .$$

We finally define the set of storage elements as:

$$SE = SEid \times Sprop \times Sz \quad .$$

A storage element is a tuple of the form

$$\langle id, sprops, size \rangle \quad .$$

5.1.3 Space

We define the following basic sets:

Space Tokens	T	a countable set of symbols
Owners	O	a finite set of symbols
Space requests	R_s	a countable set of symbols

We finally define the set of spaces as:

$$S = T \times L \times Prop \times Sz \times O \times R_s \quad .$$

A space is a tuple of the form

$$\langle token, lifetime, props, size, owner, request \rangle \quad .$$

5.1.4 Copy

We define the following basic sets:

Physical File Names	Pfn	a countable set of symbols
Copy requests	R_c	a countable set of symbols

We define the set of copies as:

$$C = Pfn \times L \times R_c \quad .$$

A copy is a tuple of the form

$$\langle physname, pintime, request \rangle \quad .$$

5.1.5 Handle

We define the following basic sets:

TURLs	Tr	a countable set of symbols
Handle requests	R_h	a countable set of symbols

We define the set of handles as:

$$H = Tr \times L \times R_h \quad .$$

A handle is a tuple of the form

$$\langle turl, pintime, request \rangle \quad .$$

5.1.6 File

We define the following basic sets:

SURLs	Sr	a countable set of symbols
File Types	Ft	$\{\mathsf{file}, \mathsf{dir}\}$
Creation Times	T_c	$\mathbb{N}$
File Storage Types	St	$\{\mathsf{volatile}, \mathsf{durable}, \mathsf{permanent}\}$
File Localities	Fl	$\{\mathsf{online}, \mathsf{online\_nearline}, \mathsf{nearline}, \mathsf{unavailable}, \mathsf{lost}\}$

For sets St and Fl we have, respectively:

$$\mathsf{volatile} < \mathsf{durable} < \mathsf{permanent}$$

$$\mathsf{online} < \mathsf{online\_nearline} < \mathsf{nearline} < \mathsf{unavailable} < \mathsf{lost} \quad .$$

We define the set of files as:

$$F = Sr \times L \times Ft \times Sz \times T_c \times St \times Sc \times Fl \quad .$$

A file is a tuple of the form

$$\langle surl, lifetime, ftype, size, ctime, stype, sclass, locality \rangle \quad .$$

5.2 Functions and relationships

After introducing sets that model entities and their properties, we can use functions to model their relationships. In particular, one-to-may relationships are modeled by functions that map to (sub)sets.

We define the following functionss:

stime Function *stime* is the *start time* of a space, copy, or handle, i.e., the time when its (pin) lifetime starts to be counted down.

$$stime : S \cup C \cup H \to \mathbb{N} \quad .$$

lleft Function *lleft* is the remaining (pin) lifetime of a space, copy, or handle at a given time.

$$lleft : (S \cup C \cup H) \times \mathbb{N} \to \mathbb{N} \quad .$$

file Function *file* gives the file owning a copy.

$$file : C \to F \ .$$

fcopies Function *fcopies* gives the set of copies of a file.

$$fcopies : F \to \mathscr{P}C \ .$$

space The space hosting a given copy.

$$space : C \to S \ .$$

scopies Function *scopies* gives the set of copies hosted by a space.

$$scopies : S \to \mathscr{P}C$$

where

$$c \in scopies(s) \Leftrightarrow c \in C \wedge s = space(c) \ .$$

refcopy Function *refcopy* gives the the copy that is referred to by a handle.

$$refcopy : H \to C \ .$$

fhandles Function *fhandles* gives the set of handles that refer to a copy of a file.

$$fhandles : F \to \mathscr{P}H$$

where

$$h \in fhandles(f) \Leftrightarrow h \in H \wedge \exists_c \, (c \in fcopies(f) \wedge c = refcopy(h)) \ .$$

shandles Function *shandles* gives the set of handles that refer to a copy held by a space.

$$shandles : S \to \mathscr{P}H$$

where

$$h \in shandles(s) \Leftrightarrow h \in H \wedge \exists_c \, (s = space(c) \wedge c = refcopy(h)) \ .$$

master A file has one *master* copy.

$$master : F \to C \ .$$

mspace The space holding a file's master copy.

$$mspace : F \to S$$

where

$$s = mspace(f) \Leftrightarrow s \in S \wedge \exists_c \, (s = space(c) \wedge c = master(f)) \ .$$

resfiles A file is *resident* on a space if the space holds the file's master copy. Function *resfiles* gives the set of files resident on a space.

$$resfiles : S \to \mathscr{P}F$$

where

$$f \in resfiles(s) \Leftrightarrow f \in F \wedge \exists_c \, s = mspace(c) \ .$$

5.3 Constraints

With the sets and functions introduced above, we can now express some of the constraints that must be satisfied by the SRM. The constraints are grouped by the main entity they refer to, and for each entity they are grouped by the main attribute or relationship affected by the constraint. Other constraints are grouped under the *Integrity* heading.

In the following, *se* denotes the storage element.

5.3.1 Space

Size The sum of all the space sizes on a storage elements cannot exceed the total available space of the storage element:

$$\sum_{s \in S} s.size \leq se.size \ .$$

Lifetimes The remaining lifetime of a space s at start time equals its assigned lifetime:

$$\forall_{s \in S}\ lleft(s, stime(s)) = s.lifetime \ .$$

Properties The properties of a space are supported by the storage element:

$$\forall_{s \in S}\ s.props \in se.sprops \ .$$

5.3.2 Copy

Integrity A copy is hosted by exactly one space:

$$\forall_{c \in C} \exists_{1 s \in S}\ c \in scopies(s) \ .$$

A copy belongs to exactly one file:

$$\forall_{c \in C} \exists_{1 f \in F}\ c \in fcopies(f) \ .$$

Pintime The pintime of a copy cannot exceed either the file's or the space's lifetime:

$$\forall_{c \in C}\ c.pintime \leq \min(space(c).lifetime, file(c).lifetime) \ .$$

The remaining pin lifetime of a copy at start time equals its assigned pinlifetime:

$$\forall_{c \in C}\ lleft(c, stime(c)) = c.pintime \ .$$

A copy cannot outlive its file:

$$\forall_{c \in C} \forall_{t > stime(c)}\ lleft(c,t) \leq lleft(file(c),t) \ .$$

5.3.3 Handle

Integrity A handle must refer to exactly one copy:

$$\forall_{h \in H} \exists_{1 c \in C}\ c = refcopy(h) \ .$$

Pintime The pintime of a handle cannot exceed the copy's pin lifetime:

$$\forall_{h \in H}\ h.pintime \leq refcopy(h).pintime \ .$$

The remaining pin lifetime of a handle at start time equals its assigned pinlifetime:

$$\forall_{h \in H}\ lleft(h, stime(h)) = h.pintime \ .$$

A handle cannot outlive its copy:

$$\forall_{h \in H} \forall_{t > stime(h)}\ lleft(h,t) \leq lleft(refcopy(h),t) \ .$$

5.3.4 File

Integrity A file resides in exactly one space:

$$\forall_{f \in F} \exists_{1 s \in S}\ f \in resfiles(s) \ .$$

Policy The storage class of a file must be supported by the storage element:

$$\forall_{f \in F}\ f.sclass \in se.sprops.sclass \ .$$

The retention policy of a file must match the space's retention policy:

$$\forall_{f \in F}\ f.sclass.retpol = mspace(f).props.sclass.retpol \ .$$

The access latency of a file must be compatible with the space's access latency:

$$\forall_{f \in F}\ f.sclass.latency \geq mspace(f).props.sclass.latency \ .$$

Lifetime The lifetime of a file cannot exceed the space's lifetime:

$$\forall_{f \in F}\ f.lifetime \leq mspace(f).lifetime \ .$$

The remaining lifetime of a file at start time equals its assigned lifetime:

$$\forall_{f \in H}\ lleft(f, stime(f)) = f.lifetime \ .$$

A file cannot outlive its space:

$$\forall_{f \in F} \forall_{t > stime(f)}\ lleft(f,t) \leq lleft(mspace(f),t) \ .$$

6 Validation of existing SRM implementations

The SRM has currently been implemented for five different Mass Storage Systems, namely:

CASTOR developed at CERN [3] and used by many other laboratories to serve data on automatic tape libraries and on disk servers used mainly as a front-end cache. The SRM 2.2 implementation for CASTOR has been made by RAL (UK).

dCache developed at DESY (Germany) [7], used by many sites with multiple MSS back-ends, both custom and proprietary. dCache can be used also as a disk-only MSS. The SRM 2.2 implementation for dCache has been made by FNAL.

DPM developed at CERN [4]. This is a disk-only based MSS. The SRM 2.2 implementation has been made at CERN.

DRM/BeStMan is the LBNL disk-based storage system. LBNL has been the first promoter of SRM. This storage system was the first prototype on which SRM has been tested.

StoRM is a disk-based system [5]. It offers an SRM 2.2 interface to parallel file systems such as GPFS or PVFS. The SRM 2.2 implementation has been made at CNAF.

All these systems have been tested for compliance with the SRM Interface Specification. Using various techniques of black-box testing [10], five families of test cases have been designed:

Availability to check the availability in time of the SRM service end-points.

Basic to verify basic functionality of the implemented SRM APIs.

Use Cases to check boundary conditions, use cases derived by real usage, function interactions, exceptions, etc.

Exhaustion to exhaust all possible values of input and output arguments such as length of filenames, SURL format, optional arguments, strings, etc.

Stress tests to stress the systems, identify race conditions, study the behavior of the system when critical concurrent operations are performed, etc.

The SRM model proposed in this paper has been used to derive several test cases in the *Basic* and *Use Cases* test suites.

7 Conclusions

A comprehensive model of the SRM has been proposed to support the development and verification of SRM implementations, using different notations and levels of formality in order to satisfy the needs of different stakeholders in the SRM development. In spite of this internal diversity, we trust that it is and will remain coherent, provided that the different levels of formality are kept separated and reciprocally consistent.

The first draft of the model is available, and feedback from its users is awaited. In fact, the model has already contributed to the validation of existing implementations by assisting in the design of a few families of tests, and its development has helped in identifying unanticipated behaviors and interactions.

The testing campaign itself has motivated the developers to reconsider many of the initial assumptions and decisions, leading to solutions that seem to better satisfy the needs of the users.

The model is still being developed with the aim to formalize the dynamic interactions, and generate test sets automatically. The current SRM implementations must be further validated for protocol compliance, since currently they do not reflect the full protocol but rather

a subset defined in the document known as *Memorandum of Understanding*, or MoU [9], an agreement on the initial SRM requirements for WLCG.

Acknowledgements The work of the many people in the SRM collaboration is gratefully acknowledged. In particular we would like to thank Arie Shoshani, Alex Sim and Junmin Gu from LBNL, Jean-Philippe Baud, Paolo Badino, Maarten Litmaath from CERN, Timur Perelmutov from FNAL, Patrick Fuhrmann from DESY, Shaun De Witt from RAL, Ezio Corso from ICTP, Luca Magnoni and Riccardo Zappi from CNAF/INFN, for their valuable input during the specification definition process. Finally, we would like to thank the WLCG project for giving us the opportunity to collect the requirements and test the proposed protocol for real use-cases.

The authors have been supported by CERN and INFN, respectively.

References

1. W. Allcock, J. Bester, J. Bresnahan, A. Chervenak, L. Liming, S. Meder, and S. Tuecke. GridFTP protocol specification. Document, GGF GridFTP Working Group, September 2002.
2. P. Badino, J.-Ph. Baud, S. Burke, E. Corso, S. De Witt, F. Donno, P. Fuhrmann, M. Litmaath, and R. Zappi. Storage Element Model for SRM 2.2 and GLUE schema description, v3.5. Technical report, WLCG, Oct. 27, 2006.
3. O. Barring, B. Couturier, J.-D. Durand, E. Knezo, and S. Ponce. Storage Resource Sharing with CASTOR. In *12th NASA Goddard/21st IEEE Conference on Mass Storage Systems and Technologies (MSST2004)*, U. of Maryland, Adelphi, MD, Apr. 13–16, 2004.
4. J.-Ph. Baud and J. Casey. Evolution of LCG-2 Data Management. In *Proceedings of the Computing in High Energy Physics (CHEP'04) conference*, Interlaken, Switzerland, September 27 – October 1, 2004.
5. E. Corso, S. Cozzini, F. Donno, A. Ghiselli, L. Magnoni, M.Mazzucato, R. Murri, P.P. Ricci, H. Stockinger, A. Terpin, V. Vagnoni, and R. Zappi. StoRM, an SRM Implementation for LHC Analysis Farms. In *Proceedings of the Computing in High Energy Physics (CHEP'06) conference*, Mumbai, India, Feb. 2006.
6. A. Domenici and F. Donno. A model for the Storage Resource Manager. In *Proceedings of the Int'l. Symposium on Grid Computing, (ISGC 2007), Taipei, Taiwan*, LNCS. Springer, 2007. In press.
7. M. Ernst, P. Fuhrmann, T. Mkrtchyan, J. Bakken, I. Fisk, T. Perelmutov, and D. Petravick. Managed data storage and data access services for data grids. In *Proceedings of the Computing in High Energy Physics (CHEP) conference*, Interlaken, Switzerland, September 27 – October 1, 2004.
8. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, July(8):231–274, 1987.
9. SRM v2.2 WLCG Usage Agreement. File `SRMLCG-MoU-day2[1].pdf` from `http://cd-docdb.fnal.gov/0015/001583/001/`, May 2006. Grid Storage Interfaces Workshop, Fermilab.
10. G. J. Myers, C. Sandler (rev. by), T. Badgett (rev. by), and T. M. Thomas (rev. by). *The Art of Software Testing*. John Wiley & Sons, 2nd edition, 2004.
11. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 2nd edition, 2004.
12. A. Shoshani, P. Kunszt, H. Stockinger, K. Stockinger, E. Laure, J.-Ph. Baud, J. Jensen, E. Knezo, S. Occhetti, O. Wynge, O. Barring, B. Hess, A. Kowalski, C. Watson, D. Petravick, T. Perelmutov, R. Wellner, J. Gu, and A. Sim. Storage Resource Management: Concepts, Functionality, and Interface Specification. In *Future of Grid Data Environments: A Global Grid Forum (GGF) Data Area Workshop*, Berlin, Germany, March 9–13, 2004.
13. A. Shoshani, A. Sim, and J. Gu. Storage Resource Managers: Middleware Components for Grid Storage. In *Proceedings of the 9th IEEE Symposium on Mass Storage Systems (MSS '02)*, 2002.
14. The Storage Resource Manager Interface Specification, Version 2.2. `http://sdm.lbl.gov/srm-wg/doc/SRM.v2.2.pdf`, April 2007. Storage Resource Manager Working Group.
15. J. Warmer and A. Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley, 2nd edition, 2004.
16. J. Woodcock and J. Davies. *Using Z – Specification, Refinement, and Proof*. Prentice Hall, 1996.